スケーラブルシステムズ株式会社

OpenMPプログラミング入門（Part 1）

講習の内容：Part 1

OpenMPによるマルチスレッドプログラミングで必要な基礎知識

- 並列プログラミングについての概要説明
  - スレッドとプロセスの違いについて
  - OpenMPと他のAPI（特にMPI)との違いについて
  - 並列化アプリケーションの開発に際してのアプローチ
- OpenMPプログラミングに関するトピックスの紹介

OpenMPプログラミング入門

並列プログラミング

並列プログラミング

- 今回のセミナーはOpenMPによるマルチスレッドプログラミングについての説明ですが、復習として、以下の点についての説明いたします。

1. **スレッドとプロセスの違いについて**
    1. **並列処理を理解するためには、プロセスとスレッドという言葉について理解することが必要です。**
2. **OpenMPと他のAPI(特にMPI)との違いについて**
3. **並列化アプリケーションの開発に際してのアプローチ**

マルチタスクと並列計算

- マルチタスク
  - 複数のタスクを同時に処理する
  - データベースやWEBなどのシステムなどでの並列処理
    - 一度に複数のユーザからの大量のデータ処理の要求
  - プロセス単位でのOSによる並列処理
    - 各タスクは複数のプロセスやスレッドを利用して処理を行う
    - プログラム自身の並列化は必ずしも必要ない
- **並列計算**
  - **特定の問題に対して、その計算処理を複数のプロセッサを利用して高速に処理する並列処理**
  - **対象となる問題を複数のコア、プロセッサを同時に利用して短時間で解く**
  - **並列プログラミングAPIを利用して、複数のプロセスとスレッドを利用するアプリケーションの開発が必要**

プロセスとスレッド

- 並列処理
  - 「同時に複数の処理（タスク）をOSが処理すること」
  - OSによる処理単位がプロセスとスレッドということになります。
- プロセス
  - OSは、要求されたタスクに対して複数のプロセスを起動してその処理を行います。
  - 複数のプロセスを利用して行う並列処理がマルチプロセスとなります。
- スレッド
  - これらのプロセス内で生成されて実際の処理を行うのがスレッドとなります。
  - プロセス内で複数のスレッドを生成して、並列処理を行うことをマルチスレッドと呼びます

プロセスとマルチプロセス

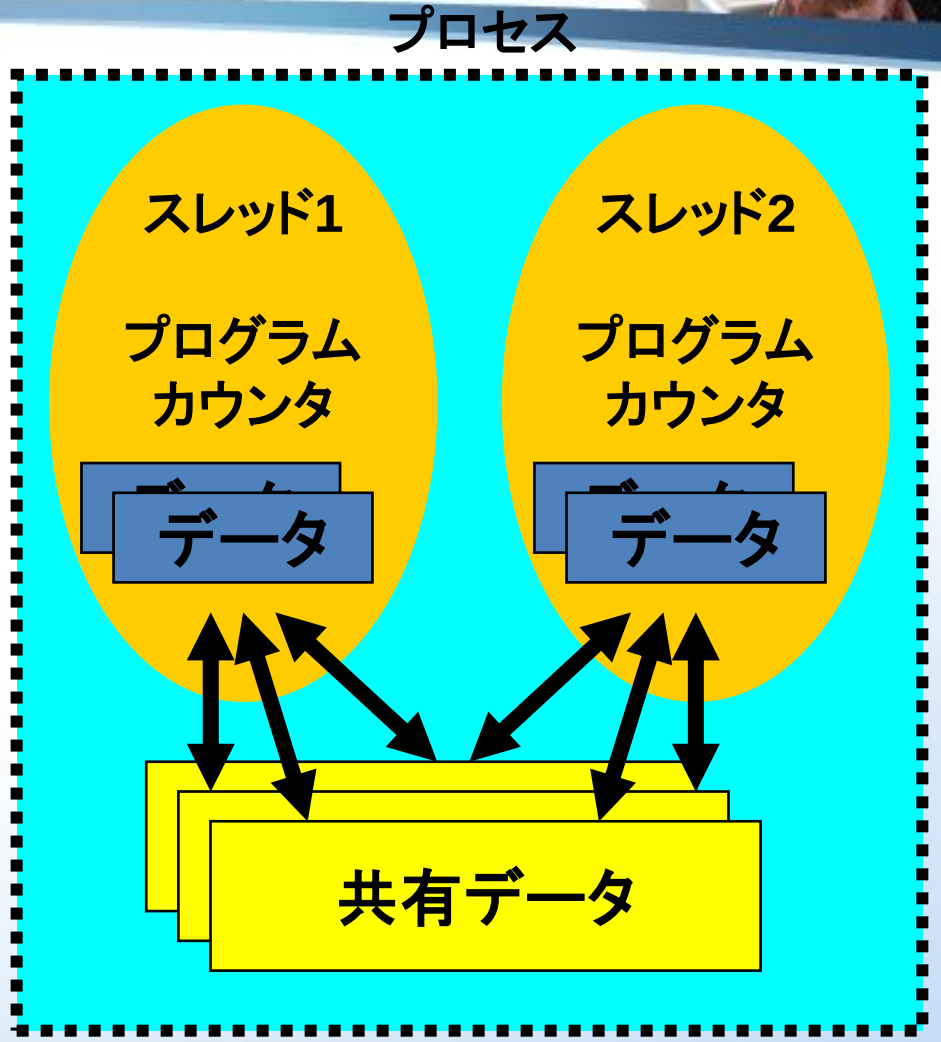
- プロセス
  - 独立した仮想メモリ空間とスタックをもち、ひとつ以上のスレッドから構成される
  - これらの仮想メモリ空間は、OSによるメモリ保護機能により、各プロセス間での干渉を防止
    - この独立した仮想メモリ空間がプロセス
- マルチプロセス
  - 複数のプロセスを利用して行う並列処理
  - プロセスの起動やそのスケジュール管理などOSが提供する様々なサービスも、複数のタスクを同時に処理するマルチプロセス
  - 複数のタスクを効率良く処理するにはマルチプロセスは最適
  - このマルチプロセスでひとつの処理・タスクを分割して複数の処理として並列に実行し、そのタスクの高速処理を行うには効率上の問題がある

マルチプロセスでの並列処理の問題

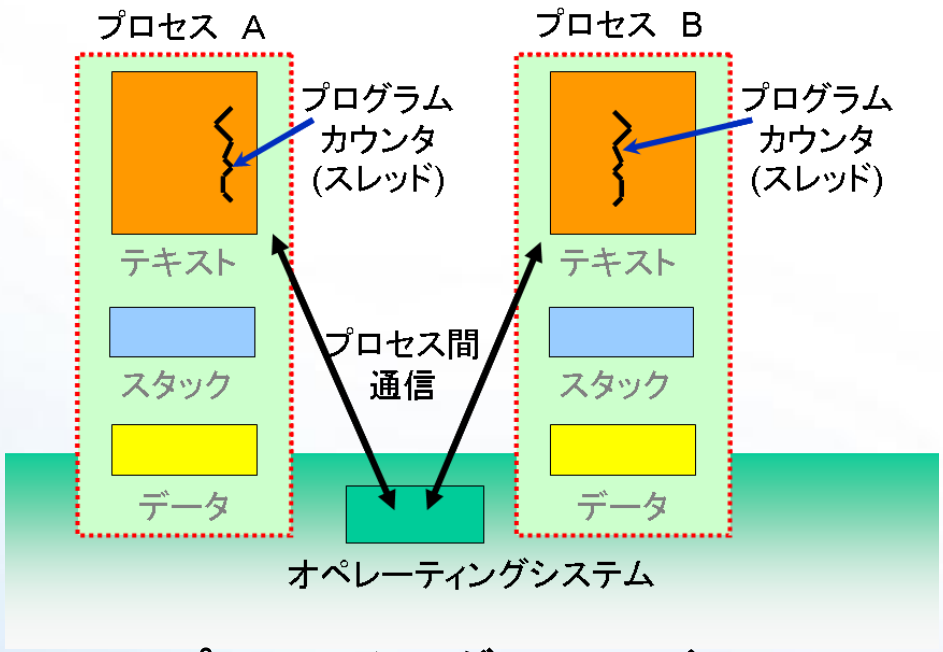
- プロセスは、メモリ保護機能によって、相互に保護される
  - プロセス間でのデータのやり取りは、すべてOSが介在したプロセッサ間でのデータ通信が必要
- 問題点
  - このようなプロセッサ間通信は、計算スピードと比較して非常に遅い処理となる
  - このプロセッサ間通信がボトルネックとなり並列処理での性能向上を十分に図ることが難しくなる
  - 発生させたプロセスについては、それらのスケジューリングと同期処理などの点で、OSの負担が大きくなる
- よりOSへの負担が少ない並列処理のリソースとして、スレッドが実装されている

プロセスとスレッドについて

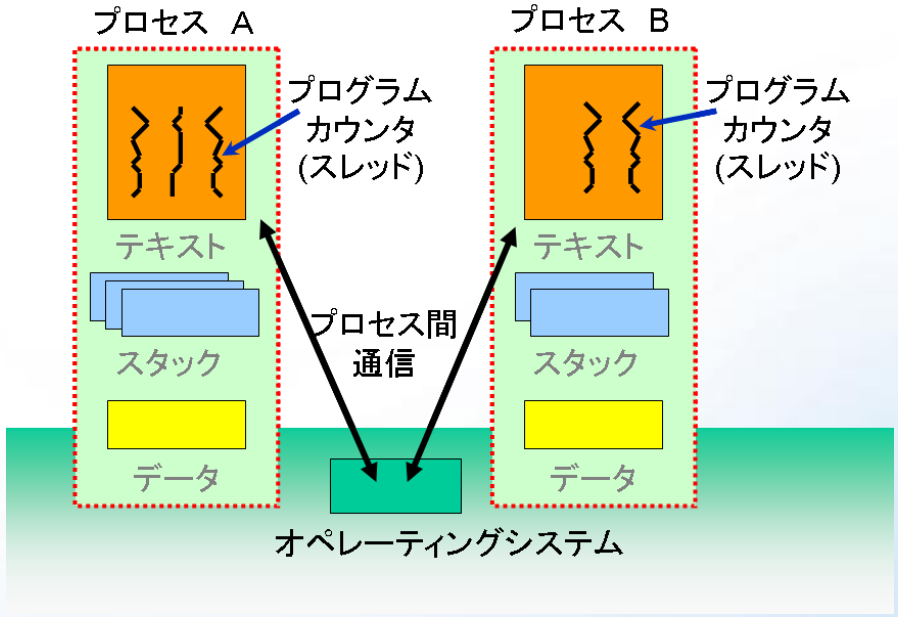
- プロセスは、独自のリソースを持って個々に実行
  - 個々に独立したアドレス空間
  - 実行命令、データ、ローカル変数スタック、ステート情報など
  - 個々のレジスタセット
- スレッドは、一つのプロセス内で実行
  - アドレス空間を共有
  - レジスタセットも共有
  - 個々のスレッドの独自データ用のスタック領域を持つ



マルチプロセスとマルチスレッド

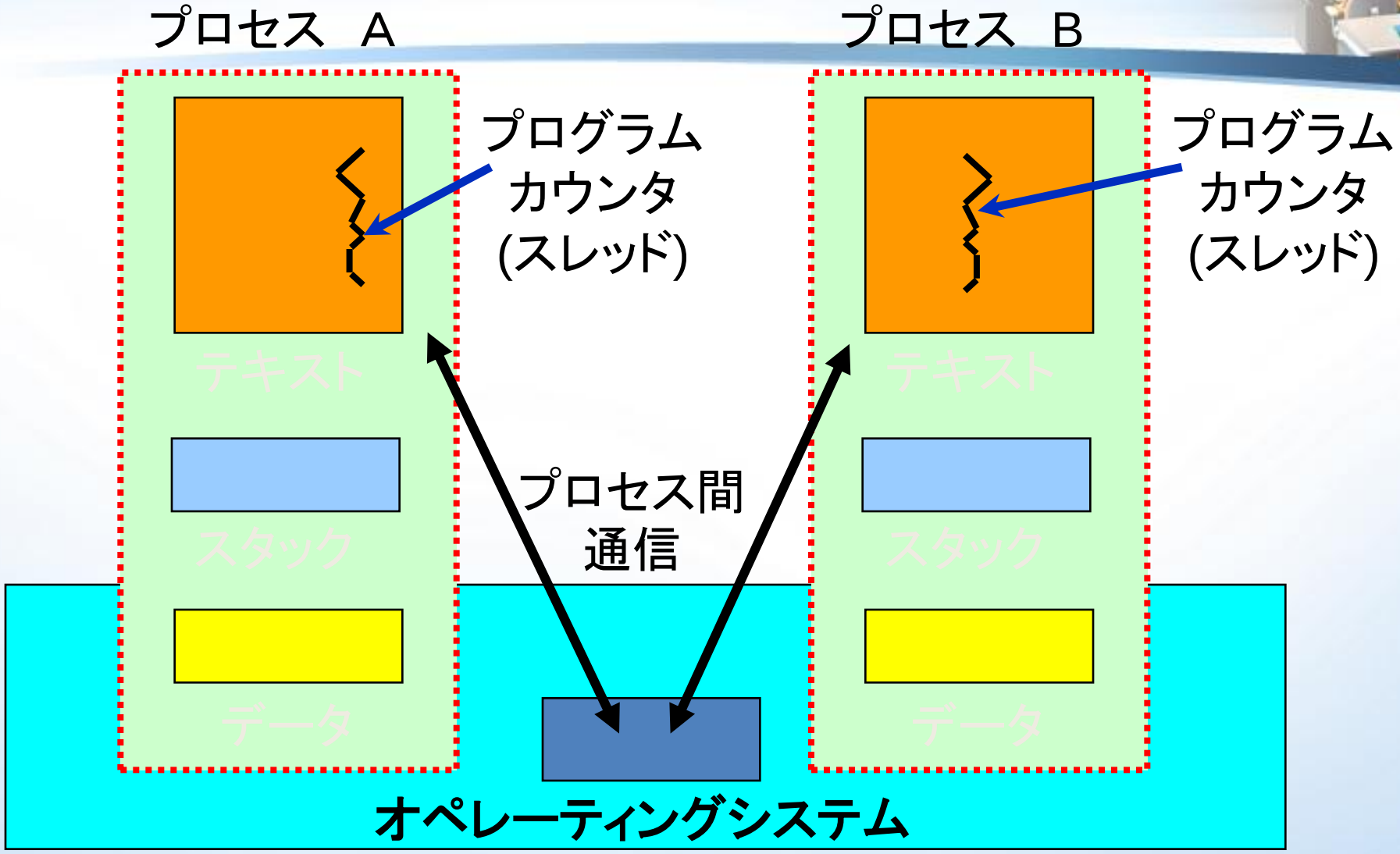


マルチプロセス/シングルスレッド



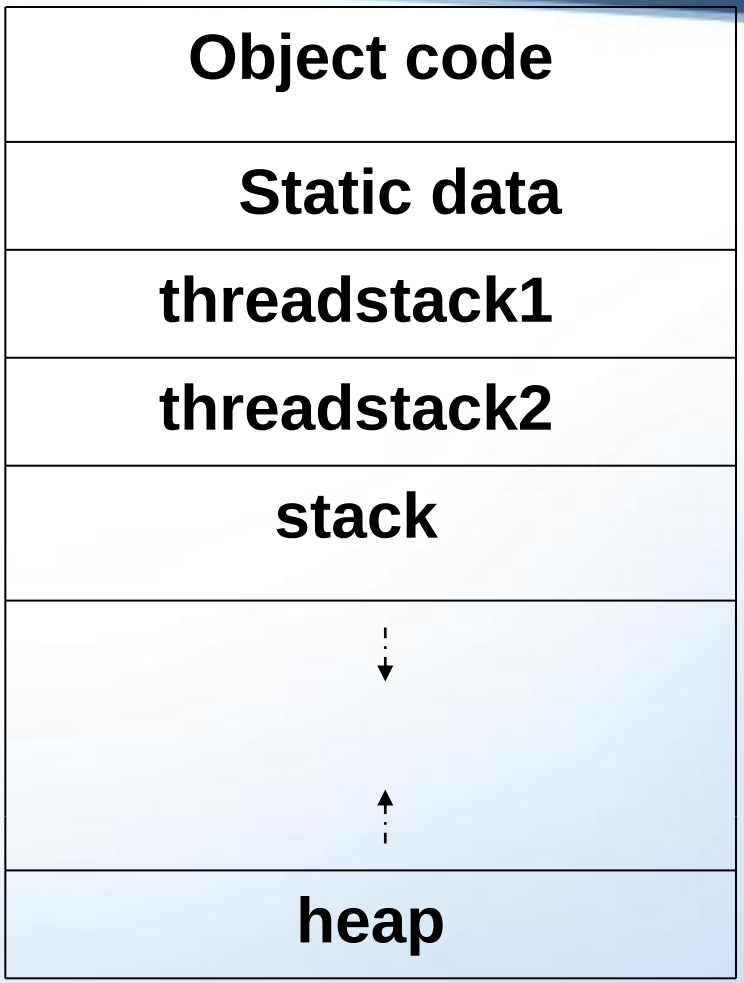
マルチプロセス/マルチスレッド

シングルスレッド、マルチプロセス

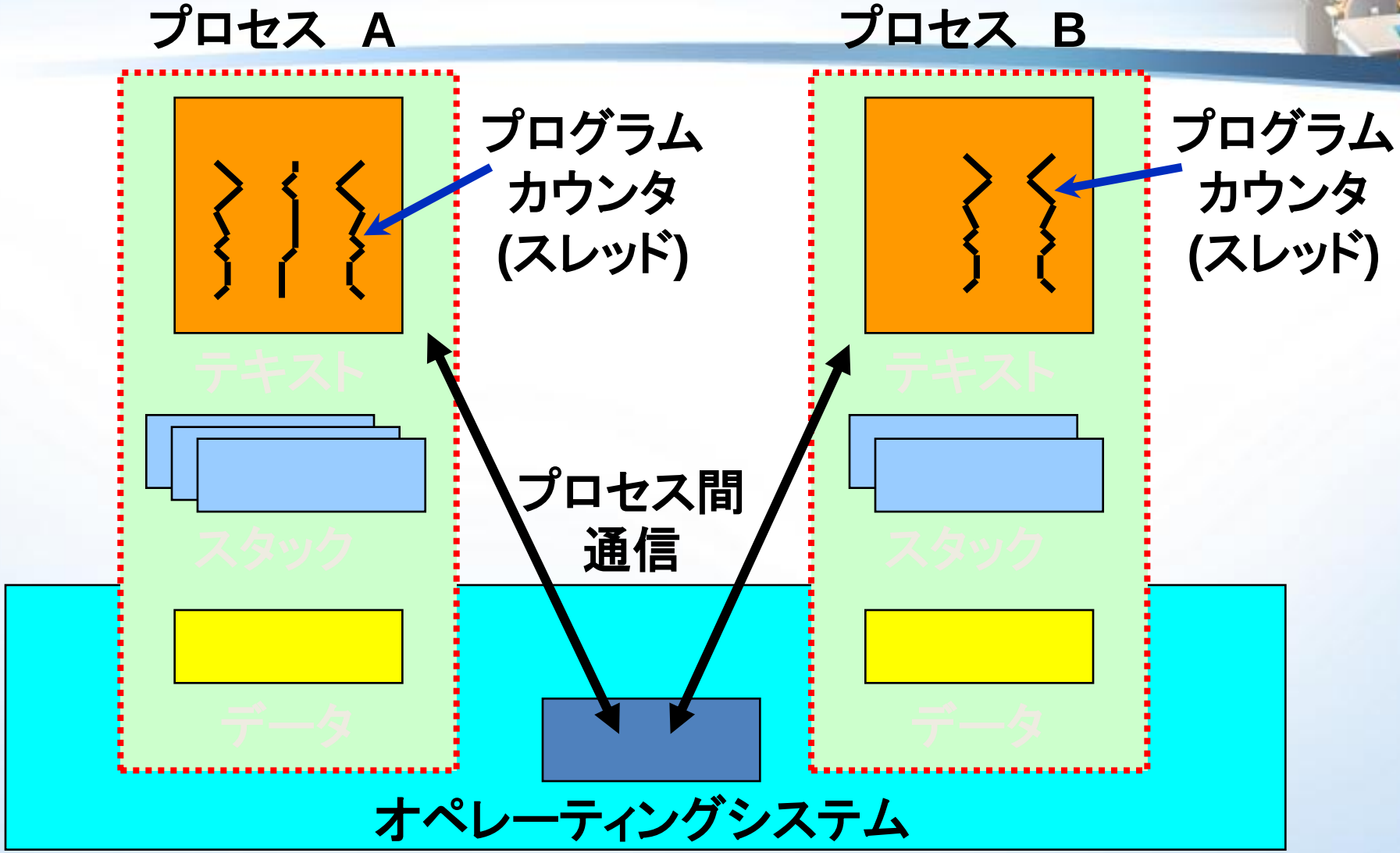


実行時のメモリ配置

- 各プロセスのメモリは、コンパイルとリンク時に決定される
- 一般的には、オブジェクトコード、グローバルデータ、ヒープ領域、プロセススタックから構成される（ヒープ領域は、言語や処理系で取り扱いが違う）
- OpenMPは、スレッドスタックが必要



マルチスレッド、マルチプロセス



プロセスの特徴

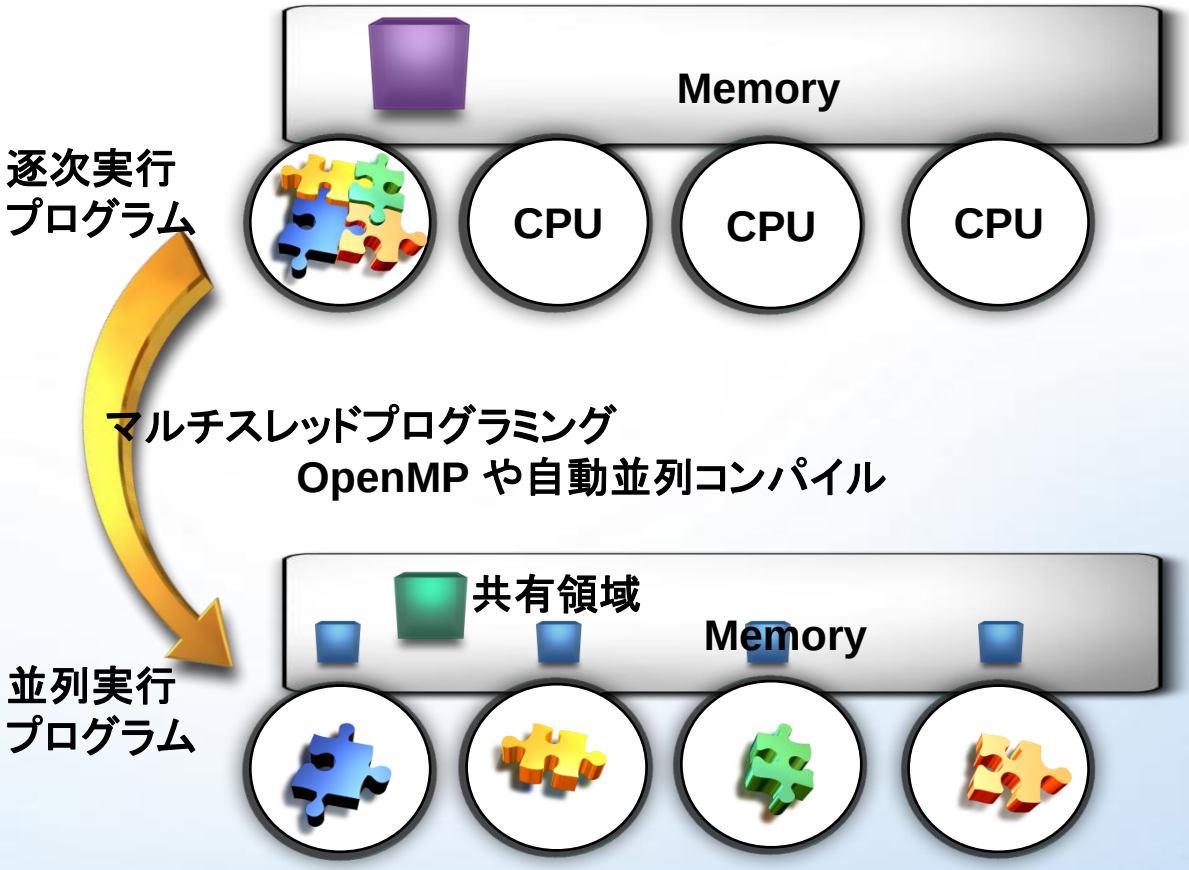
- プロセスは、‘OSに対しての負荷が大きい – Heavy Weight’
- プロセスは、独立した仮想メモリ空間とスタックをもつ
- ひとつ以上のスレッドから構成
- 仮想メモリ空間は、OSによるメモリ保護機能により、各プロセス間での干渉が防止
- プロセスの起動やそのスケジュール管理などは全てOSが提供する
- 親プロセスからの子プロセスの起動などは、自身のプログラムのロードやメモリの確保、初期化などの作業を伴う

スレッドの特徴

- プロセス内の仮想メモリ空間といったリソースやコンテキストを共有し、固有のスタックとプログラムカウンター(PC)を個別に持つ
- メモリ空間の共有により、スレッド間でのデータのやり取りは直接アクセスが可能となり、OSを介した通信のようなオーバヘッドの大きなオペレーションを必要としない
- スレッドの生成や切り替えはプロセスの場合と比較して高速

共有メモリプログラミング

スレッドを活用したのが、OpenMPによるマルチスレッドプログラミングです。OpenMPは、共有メモリ上でのプログラミングとなりますので、次に、共有メモリプログラミングとOpenMPの特徴を簡単にご説明いたします。



並列計算

- プログラム中には、多くの並列処理可能な処理が存在しているが、通常はそれらの処理を逐次的に処理している
- これらの並列処理可能なコードセグメントに対して、複数のプロセッサ(コア)による同時・並列処理を行う

タスク並列処理:
独立したサブプログラムの並列に呼び出す

```
call fluxx(fv,fx)
call fluxy(fv,fy)
call fluxz(fv,fz)
```

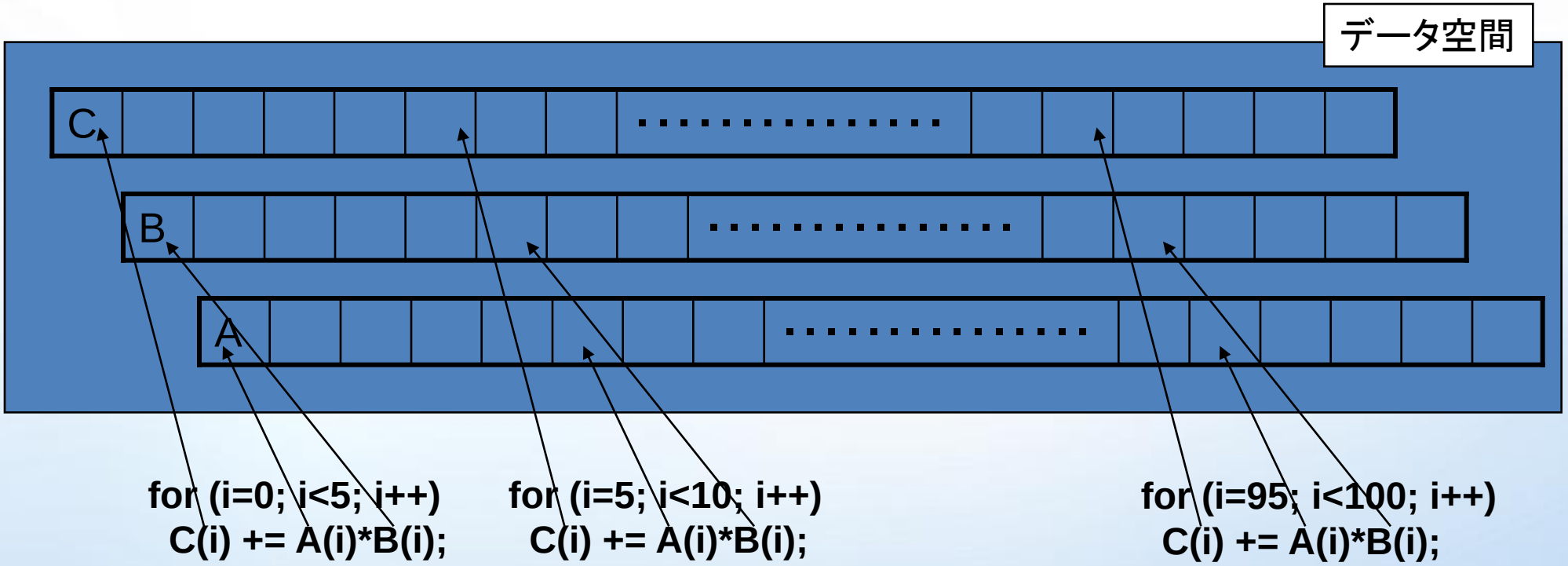
データ並列処理:
独立したループ反復を分割し、並列に実行する

```
for (y=0; y<nLines; y++)
   genLine(model,im[y]);
```

共有メモリデータ並列処理

- 並列処理の一つの方式
- データ空間を共有して並列化を行う

```
for (i=0; i<100; i++)
  C(i) += A(i)*B(i);
```



マルチスレッドプログラミングの基本

- 計算負荷の大きなループやプログラムのセクションを複数のスレッドで同時に処理
- 複数のスレッドを複数のプロセッサコア上で、効率良く処理する

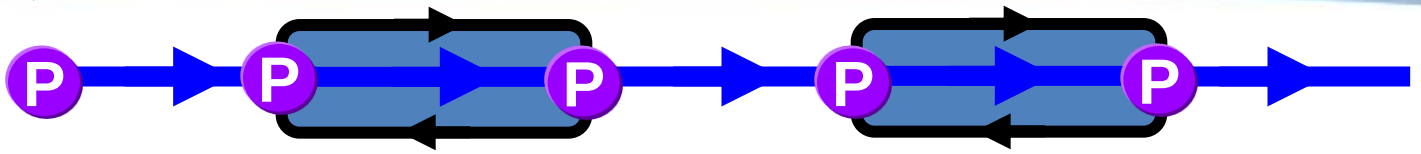
```
void main()
{
    double Res[1000];
// 計算負荷の大きな計算ループに対して、
// マルチスレッドでの並列処理を適用します
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

OpenMP の適用

```
void main()
{
    double Res[1000];
    #pragma omp parallel for
    for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
    }
}
```

逐次処理 .vs. マルチスレッド並列処理

逐次処理

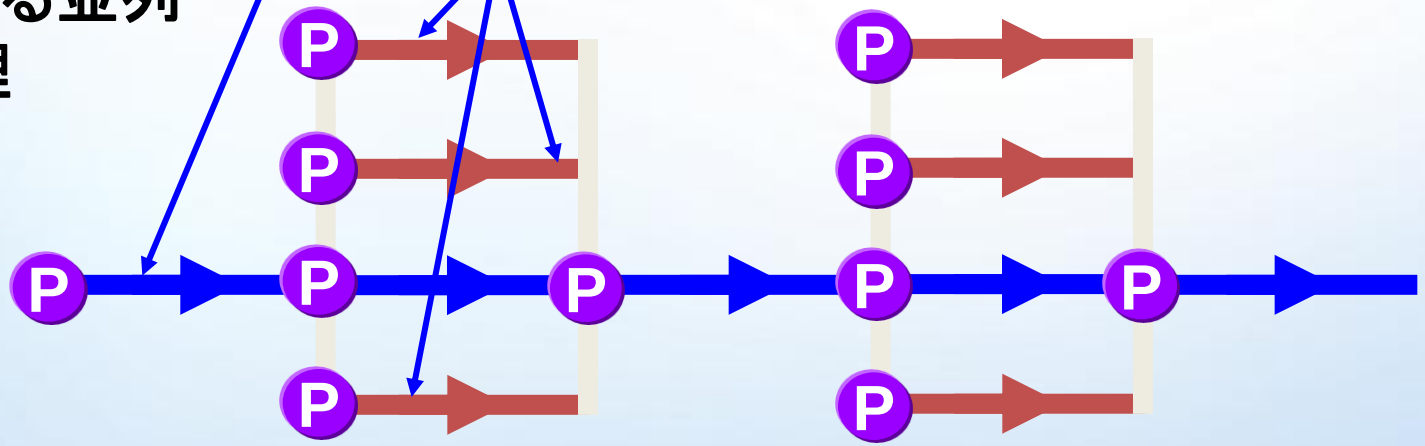


プログラムのループなどの反復計算を複数のスレッドに分割し、並列処理を行う

“マスタースレッド”

“ワーカースレッド”

マルチスレッドによる並列処理



並列モデルの比較

	MPI	スレッド	*OpenMP*
可搬性	✓		✓
スケーラブル	✓	✓	✓
パフォーマンス指向	✓		✓
並列データのサポート	✓	✓	✓
インクリメンタル並列処理			✓
高レベル			✓
直列コードの保持			✓
正当性の確認			✓
分散メモリ	✓		*ClusterOpenMP*

Win32 APIによるπの計算

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
  int i, start;
  double x, sum = 0.0;
  start = *(int *) arg;
  step = 1.0/(double) num_steps;
  for (i=start;i<= num_steps; i=i+NUM_THREADS){
       x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
          thread_handles[i] = CreateThread(0, 0,

           (LPTHREAD_START_ROUTINE) Pi,
                    &threadArg[i], 0, &threadID);
}
  WaitForMultipleObjects(NUM_THREADS,
           thread_handles,TRUE,INFINITE);

  pi = global_sum * step;
  printf(" pi is %f ¥n",pi);
}
```

MPIによるπの計算

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD) ;
}
```

OpenMPによるπの計算

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{       int i; double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
for (i=0;i<= num_steps; i++){
    x = (i+0.5)*step;
    sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
}
```

Cluster OpenMPによるπの計算

```
#include <omp.h>
static long num_steps = 1000000; double step;
static double sum = 0.0;
#pragma intel omp sharable(sum)
#pragma intel omp sharable(num_steps)
#pragma intel omp sharable(step)
#define NUM_THREADS 4
void main ()
{
        int i; double x, pi;
        step = 1.0/(double) num_steps;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=0;i<= num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

MPIとOpenMPのAPIとしての比較

	MPI （メッセージパッシング）	OpenMP
利点	• 分散メモリシステムと共有メモリシステムの双方で利用可能 • ノードサイズを超えての並列処理が可能 • データ配置の制御が容易	• 並列化が容易 • 低いレイテンシと高いバンド幅 • 通信制御が不要 • 粒度に依存しない並列化が可能（細粒度と疎粒度の双方が可能） • 動的な負荷分散（ロードバランス）が可能
問題点	• プログラム開発が容易でなく、また、デバッグが困難 • 高いレイテンシと低いバンド幅 • 疎粒度でのプログラミングが必要（ループレベルでの並列化は難しい） • 負荷分散（ロードバランス）が難しい	• 共有メモリシステムだけ • ノードサイズがスケーラビリティの限界 • データ配置が問題になる可能性がある • スレッドの細かな制御が困難

MPIとOpenMPのAPIとしての比較

	MPI （メッセージパッシング）	OpenMP
並列化	• 疎粒度での並列化 • 一般には、SPMD型のプログラミング • データ並列でもタスク並列でも利用可能	• 疎粒度での並列化も可能 • 一般には、ループレベルでの並列化を行うが、SPMD型のプログラミングも可能 • データ並列でもタスク並列でも利用可能 • OpenMPの基本は、スレッドのワークシェアであるが、個々のスレッドへのデータのアサインも可能
	• 複数のプロセスから構成される • ‘Shared Nothing’ プロセス • 陽的なメッセージ交換 • 同期処理はメッセージ交換時に実行される	• 複数スレッドから構成される • スレッドスタック以外は、全て共有される • 陽的な同期処理が必要 • 共有データへのアクセス

データの共有と保護

	メッセージ・パッシング	スレッド
データの共有	• メッセージを送受信 • ブロードキャスト • スキャッタ、ギャザ	• 共有メモリ領域に値を格納
データの保護	• 別のプロセスからメモリを読み取ることができない	• スレッドローカル格納領域 • スレッドスタックと関数からのスコープ • OpenMP* スコープ • ミューテックス（Mutex）
データの競合		• 複数のスレッドが共有データにアクセス • 実行順は仮定されているが保証されていない • 診断が困難

OpenMP SPMD スタイル

- SPMD → Single Program Multiple Data
- MPIでの並列化と同様にループレベルではなく、広範囲な領域での並列処理の検討
- ワークシェアリングだけでなく、データも各スレッドに分散することになる
  - 配列などを分割して、各スレッド用のローカル配列とする
  - グローバル配列へのアクセスを少なくして、各スレッドはスレッド・プライベートなデータを利用しての計算を行う
- MPIでのランクの設定と同じように、OpenMPでのスレッド番号を利用して、並列処理の制御を行う
  - プログラムとしては、MPIに近いが、計算のコアの部分で使うデータだけ、プライベート配列にする

数値演算手法

- 構造的で定型的なアプリケーション
  - 並列化の適用のための検討は比較的容易
  - OpenMPでもMPIでの並列化が可能

- 非構造で実行が定型的でないアプリケーション
  - 並列化の適用の検討が容易でない
  - MPIでの並列化については、プログラムのアルゴリズムやデータ構造に関する情報や知識が必要
  - OpenMPでの並列化は比較的容易に可能であるが、同期処理でのオーバーヘッドが大きい

100の柵の塗装

100の柵の塗装

- 準備: 塗料缶、ブラシを用意する、その他（～1 時間）
- 塗装: 1 柵 = 6 分（0.1 時間）
- 後片付け: ブラシを掃除する、塗料缶に蓋をする、その他（～1 時間）

人数	1	10	100
準備（時間）	1	1.1	1.2
塗装（時間）	10	1	.1
後片付け（時間）	1	1.1	1.2
処理時間合計	12	3.2	2.5
スピードアップ	1	3.75	4.8

プロセッサ数

逐次処理

並列処理

逐次処理

S = 逐次処理 / 並列処理

並列計算の問題点
十分な並列度：Amdahl's Law

- アプリケーションの一部だけが並列化できると仮定した場合のスピードアップ
- アムダールの法則：Amdahl's law
  - プログラム中での比率でsが、逐次実行されるとすると、（1－s）が並列に実行される
  - P を並列に実行するプロセッサ数とすると、スピードアップは以下のような式となる

$$\text{Speedup}(P) = \text{Time}(1)/\text{Time}(P)$$
$$\leqq 1/(s + (1-s)/P)$$
$$\leqq 1/s$$

  - プロセッサ数に応じて、完全な並列性が得られたとしても、アルゴリズムのシリアル部分が、並列実行のスピードアップを制限する

アムダールの法則

$$\text{並列化効率} = \frac{1}{p + N(1-p)} \qquad \cdots(1)$$

$$Speedup = \text{並列化効率} \times \mathrm{N} = \frac{1}{(1-p) + p/N} \qquad \cdots(2)$$

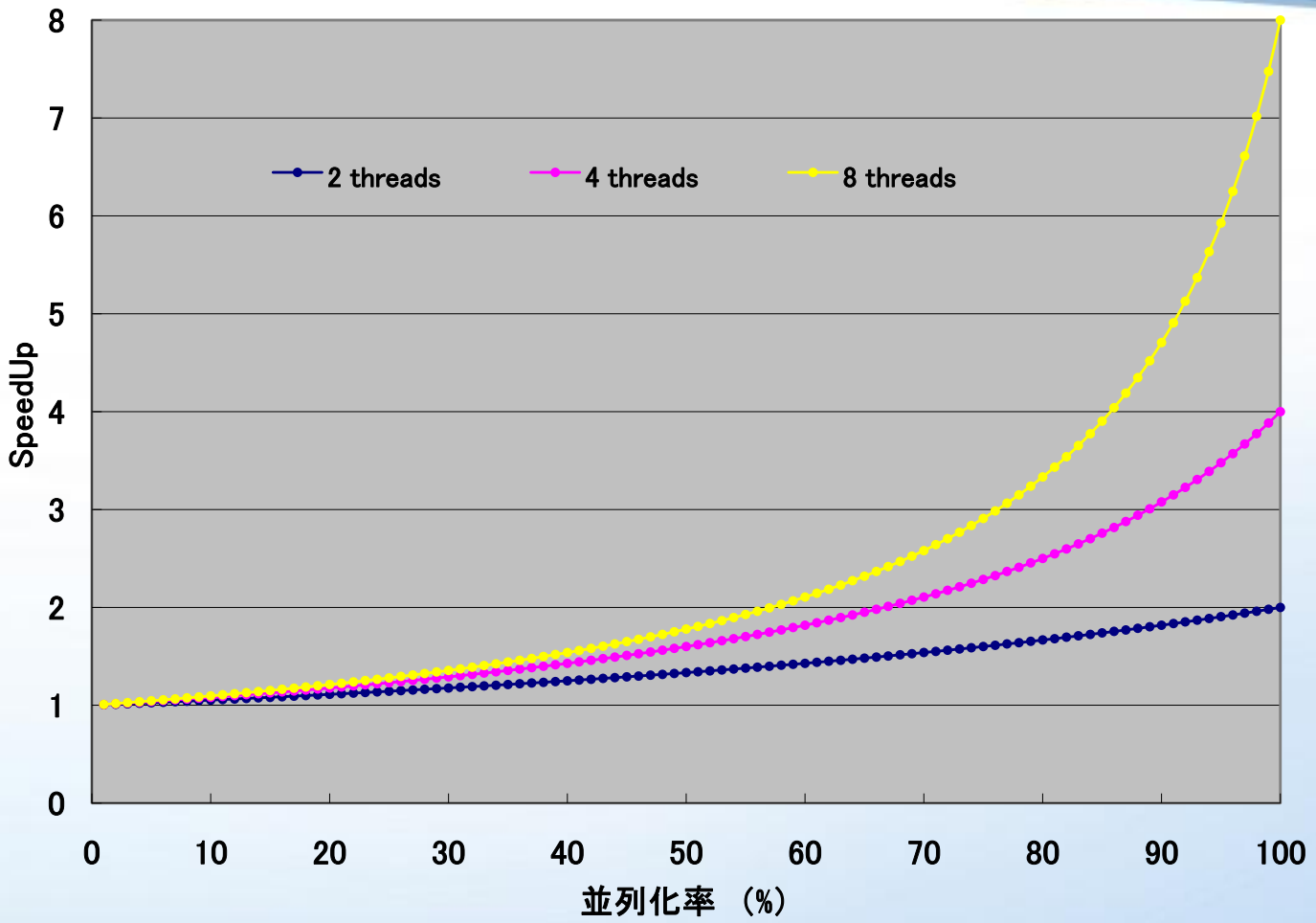
$\mathrm{N} =$ プロセッサ数

$$\text{並列化率}(\mathrm{p}) = \frac{\text{並列化可能部分のシングルスレッドでの実行時間}}{\text{シングルスレッドでの実行時間}}$$

アムダールの法則

- 並列化効率　ε

$$\varepsilon = \frac{1}{p + N(1-p)}$$



並列計算の問題点
ロード・バランシング

- ロード・バランシング: 並列コンピュータのプロセッサ間の作業の分配

「最も時間のかかる作業が終わるまで、全体の作業は完了しない」

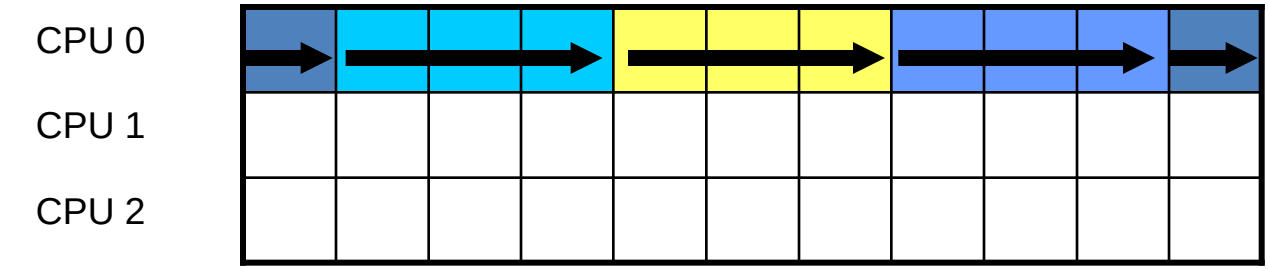
  - スタティック・ロード・バランシング: 分配は、プログラムのスタートアップ時に決定されセットアップされる
  - ダイナミック・ロード・バランシング: 分配は、計算の進行に伴って変更される
- 総合的なパフォーマンスは、最も時間がかかるプロセッサに依存する
  →最高のパフォーマンスを得るには、プロセッサにすべて均等に負荷を与える必要がある

並列計算の問題点
粒度

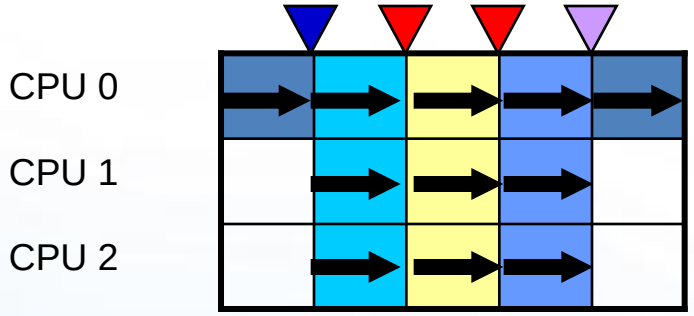
- アプリケーションの粒度 = 計算負荷/通信負荷
  - 粗粒度: レンダリング、パラメータ解析など、各タスク間での通信をほとんど必要としないアプリケーション
  - 細粒度: 計算流体力学、構造分析など、計算時に多くの通信が発生するアプリケーション
- ハードウェアの粒度 = 計算能力/通信能力
  - 粗粒度: 高速イーサネット相互接続のクラスタ
  - 中粒度: Infiniband相互接続のクラスタ
  - 細粒度: SMP システム

計算粒度：細粒度と疎粒度での並列処理

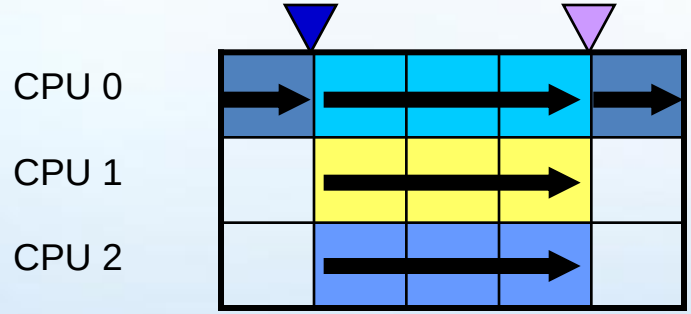
逐次処理



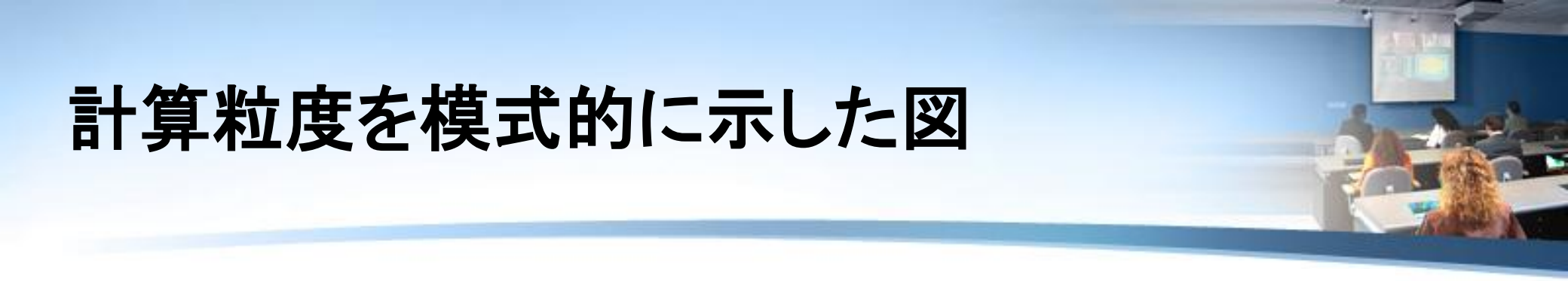
並列処理：細粒度での並列処理



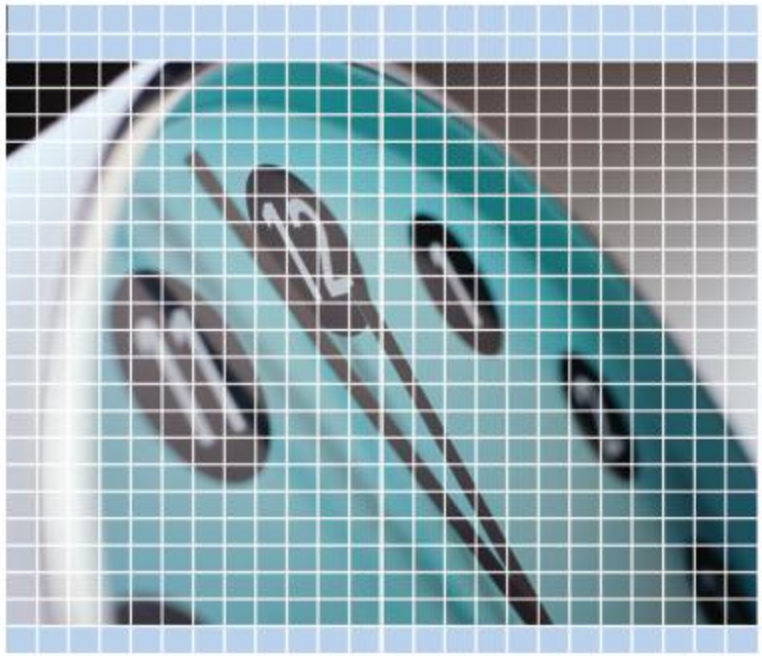
並列処理：疎粒度での並列処理



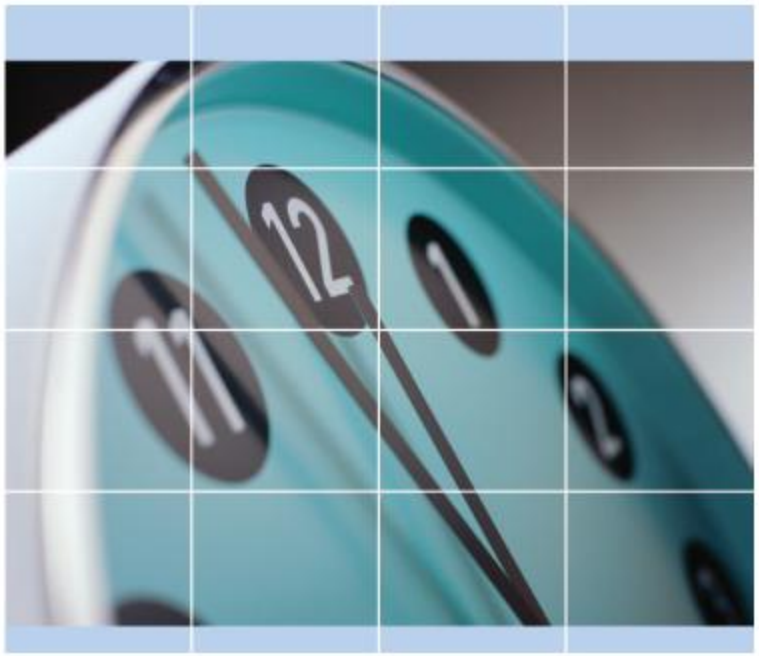
計算粒度を模式的に示した図



細粒度での領域分割



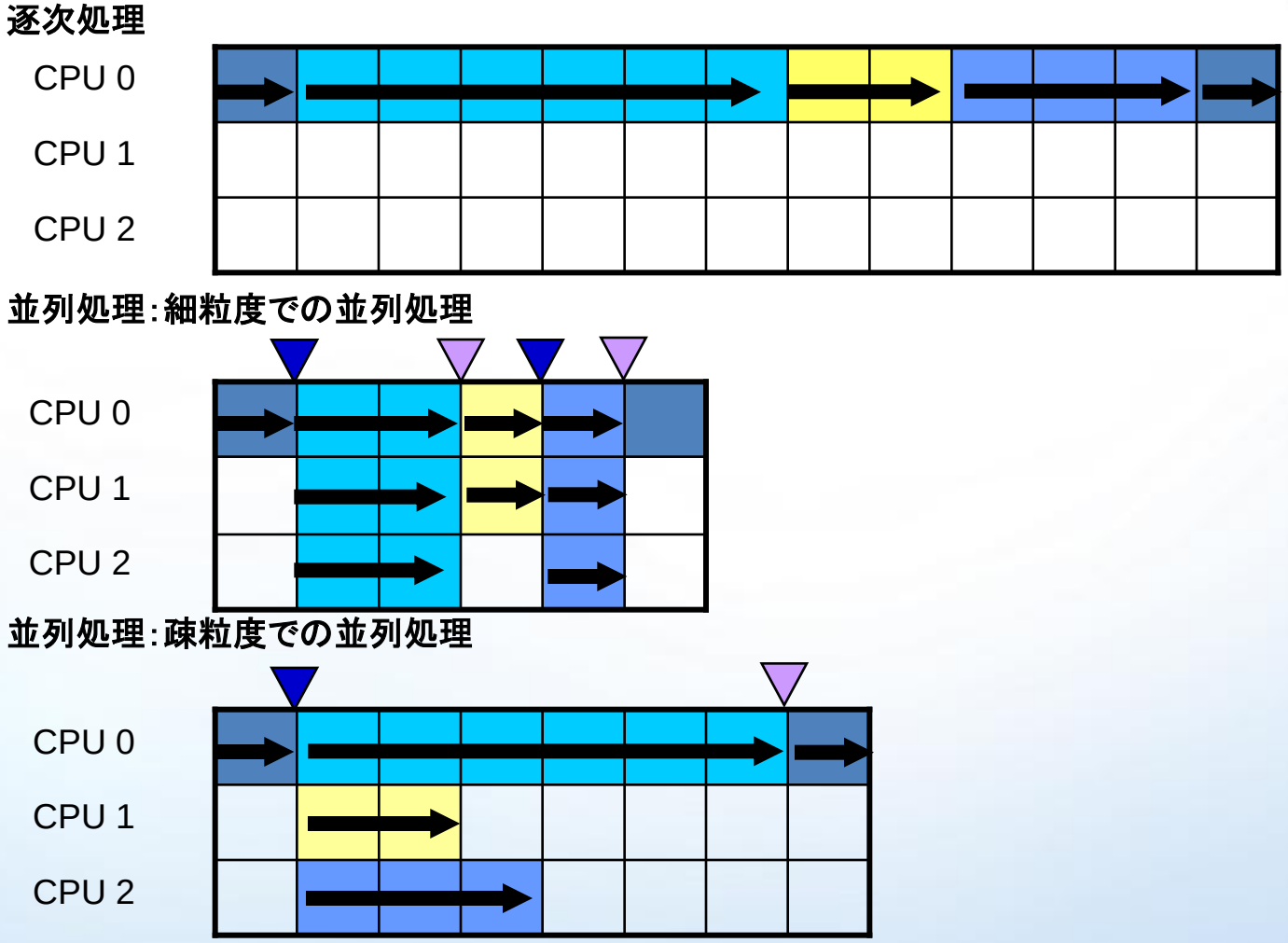
疎粒度での領域分割



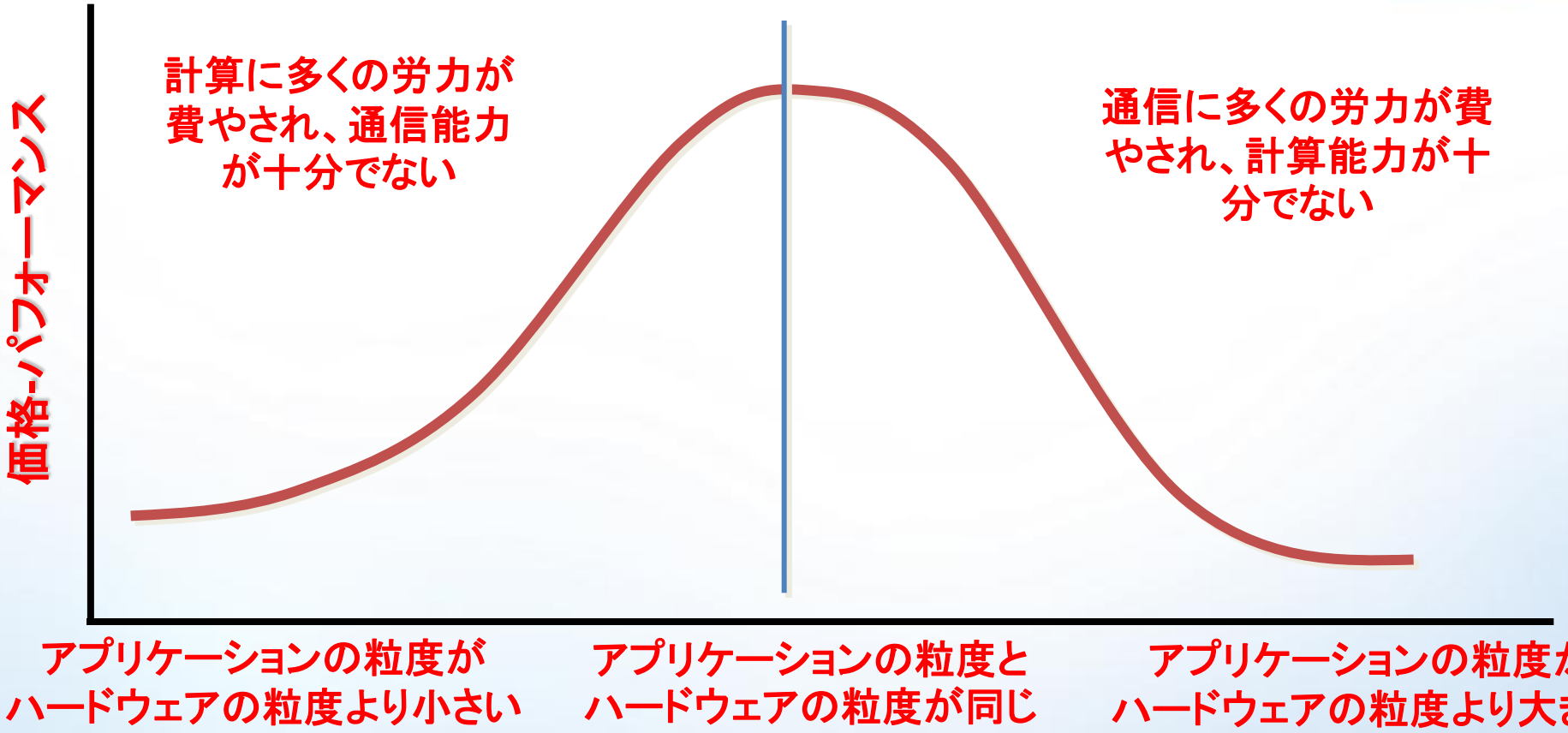
計算粒度を模式的に示した図：領域を例えばスレッド数に分割し、それぞれの領域を個々のスレッドが計算するような場合は疎粒度での領域分割となります。一方、領域を細かく分割し、各スレッドが複数の領域を順次処理するような場合は細粒度での領域分割となります。

インテル® コンパイラ
OpenMP* 入門
デュアルコア/マルチコア対応アプリケーション開発①より

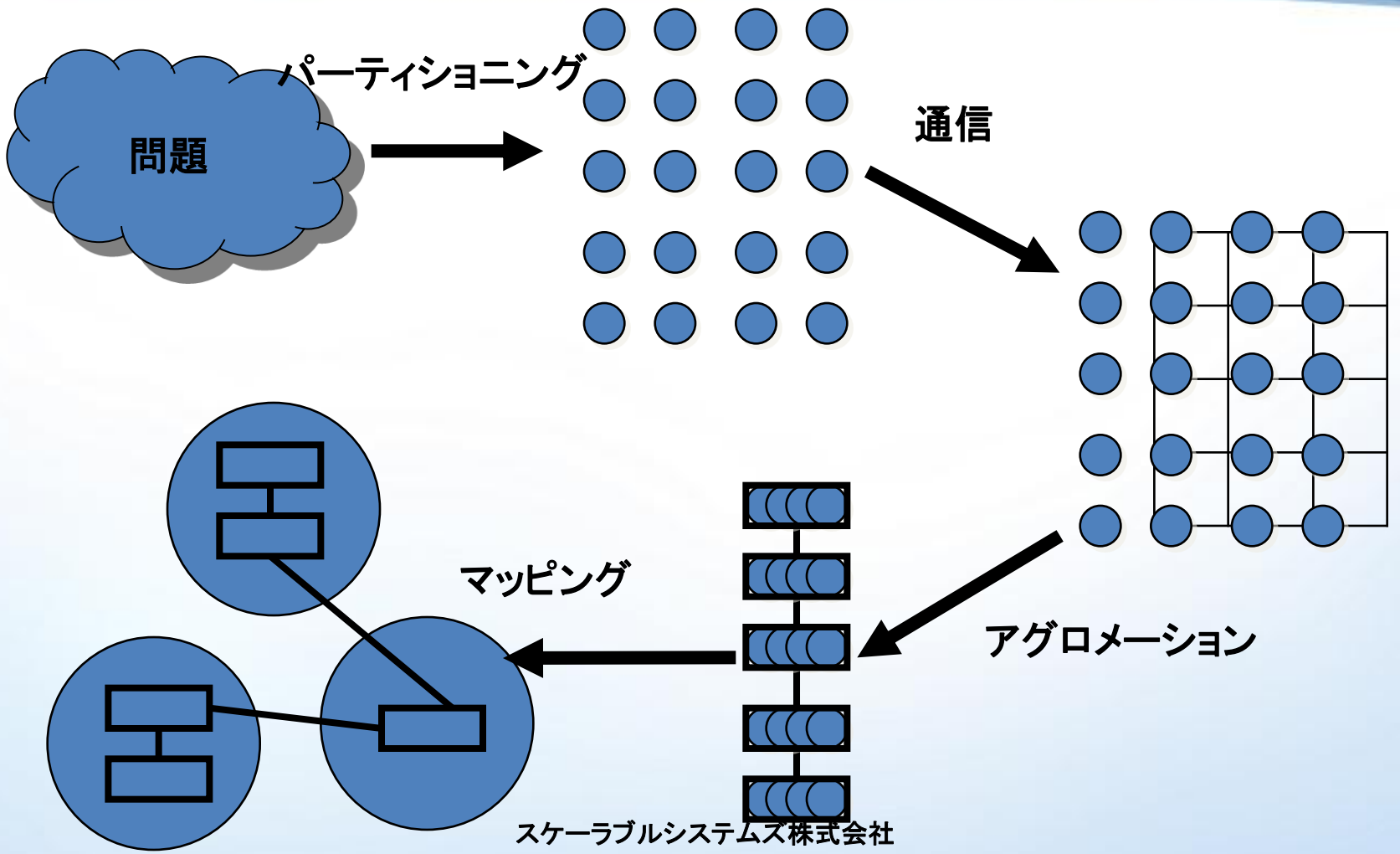
計算粒度とワークロードの分散



並列計算の問題点
アプリケーション vs. ハードウェア



Ian Fosterの設計方法論

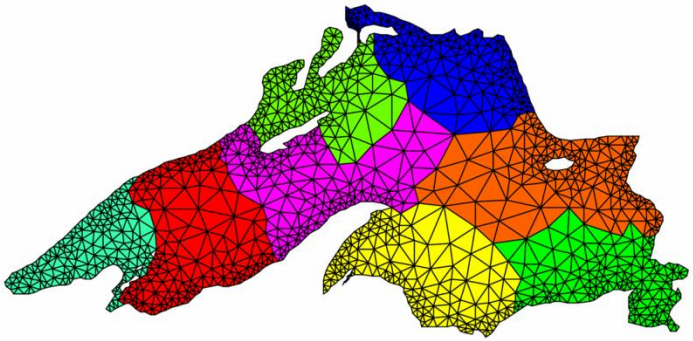


Ian Fosterの設計方法論
Designing and Building Parallel Programs

4つのステップ

1. パーティショニング
   - 演算とデータを分割
   - 問題をタスクに分割
2. 通信
   - 演算間のデータを共有
   - 通信の量とパターンを決定
3. アグロメレーション（結合）
   - タスクをグループ化してパフォーマンスを向上
   - タスクを結合
4. マッピング
   - プロセッサ/スレッドにタスクを割り当て
   - 結合されたタスクを物理プロセッサに割り当て

パーティショニング



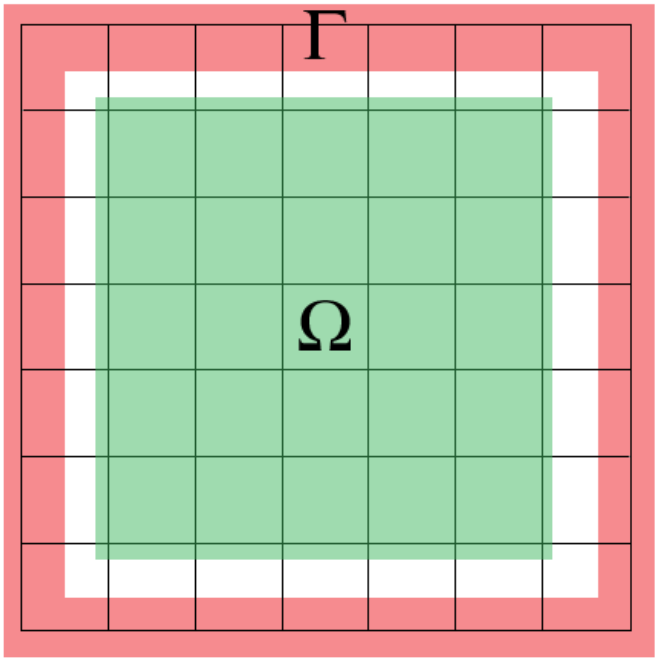
- 計算処理とデータをより小さな処理単位とデータに分割
- 領域分割（Domain decomposition）
  - データの細分化
  - 細分化したデータに対する処理の相互関係に関しての検討が必要
- 機能分割（Functional decomposition）
  - 計算処理の細分化
  - 分割された計算処理でのデータの取り扱いに関しての検討が必要

並列化事例
熱伝導方程式

- 以下のポアソン方程式の解法

$$\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}=f(x,y),0\le x\le a,0\le y\le b$$

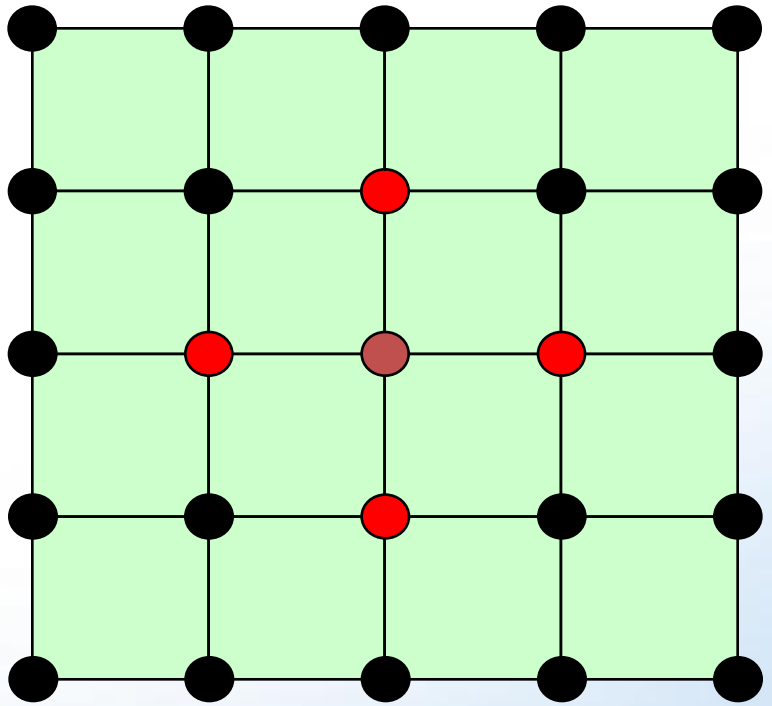
- 境界条件
  - $u(x,0)=G_1(x)$ $u(x,b)=G_2(x)$ $0 \leqq x \leqq a$
  - $u(0,y)=G_3(y)$ $u(a,y)=G_4(y)$ $0 \leqq y \leqq b$



熱伝導方程式 –ポアソン方程式

- ポアソン方程式のヤコビ法での処理

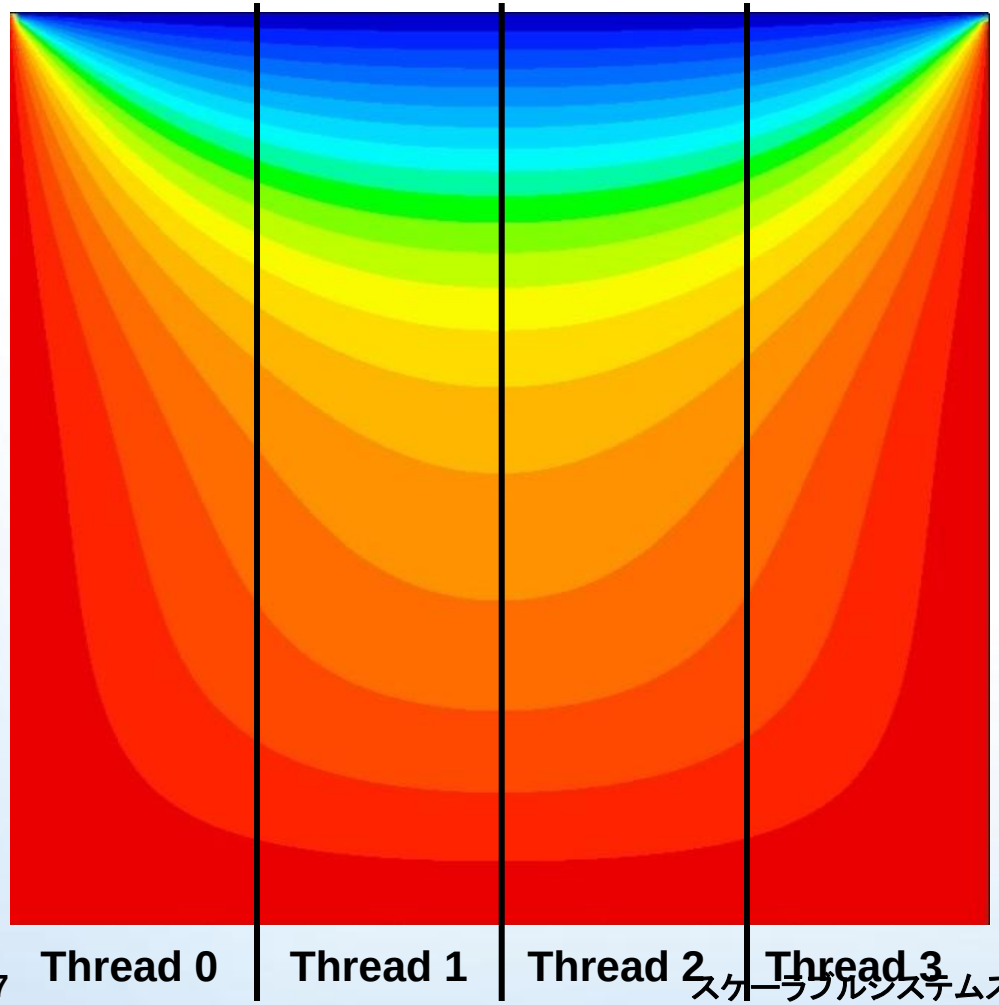
$$w_{i,j} = \frac{w_{i+1,j} + w_{i-1,j} + w_{i,j+1} + w_{1,j-1}}{4}$$



- w[i][j]の数値は、u[i-1][j]、u[i][j+1]、u[i+1][j]、u[i][j-1] の数値から計算される

```
for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++) {
          w[i][j] = (u[i-1][j] + u[i+1][j] +
                    u[i][j-1] + u[i][j+1])/4.0;
```

熱伝導方程式 –ポアソン方程式




```
float w[*][*], u[*][*];

Initialize u;
while (!stable) {
    copy w => u;  //swap pointers
    for i = …
        for j = …
            Compute w;
    for i = …
        for j = …
            Sum up differences;
    stable = (avg diff < tolerance);
}
```

熱伝導方程式 –ポアソン方程式

```
/* Sequential Solution to Steady-State Heat Problem */
#define N       10
#define EPSILON 0.01
int main (int argc, char *argv[])
{
  double diff;    /* Change in value */
  int    i, j;
  double mean;    /* Average boundary value */
  double u[N][N]; /* Old values */
  double w[N][N]; /* New values */
/* set boundary values and compute mean boundary value */
   mean = 0.01;
   for (i = 0; i< N; i++) {
      u[i][0] = u[i][N-1] = u[0][i] = 100.0;
      u[N-1][i] = 0.0;
      mean += u[i][0] + u[i][N-1] + u[0][i] + u[N-1][i];
   }
   mean /= (4.0 * N);
/* Initialize interior values */
   for (i = 1; i < N-1; i++)
      for (j = 1; j < N-1; j++) u[i][j] = mean;
```

```
/* Compute steady-state solution */
   for (;;) {
      diff = 0.0;
      for (i = 1; i < N-1; i++)
         for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > diff)
                     diff = fabs(w[i][j]-u[i][j]);
         }
         if (diff <= EPSILON) break;
         for (i = 1; i < N-1; i++)
            for (j = 1; j < N-1; j++) u[i][j] = w[i][j];
   }
 /* Print solution */
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++)
         printf ("%6.2f ",u[i][j]);
         putchar ('¥n');
   }
}
```

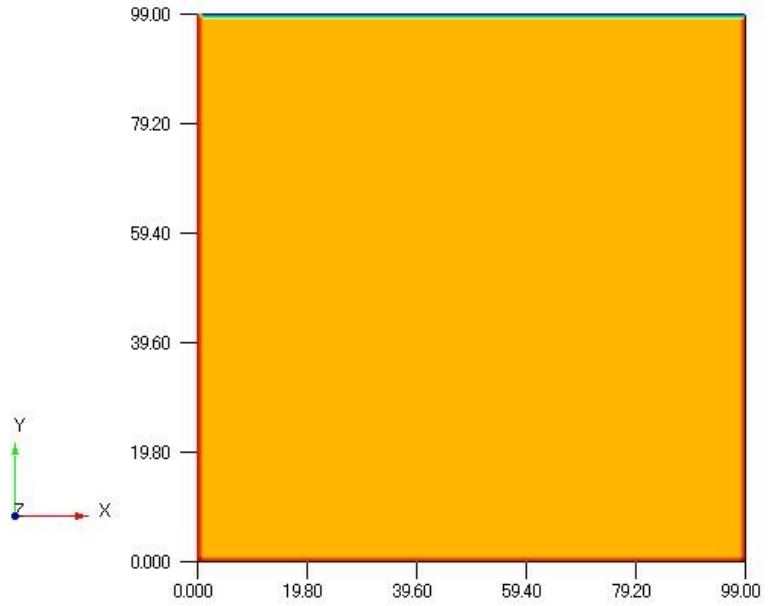
熱伝導方程式 –ポアソン方程式

```
/* Sequential Solution to Steady-State Heat Problem */
#define N       10
#define EPSILON 0.01
int main (int argc, char *argv[])
{
  double diff;    /* Change in value */
  int    i, j;
  double mean;    /* Average boundary value */
  double u[N][N]; /* Old values */
  double w[N][N]; /* New values */
/* set boundary values and compute mean boundary value */
   mean = 0.01;
   for (i = 0; i< N; i++) {
      u[i][0] = u[i][N-1] = u[0][i] = 100.0;
      u[N-1][i] = 0.0;
      mean += u[i][0] + u[i][N-1] + u[0][i] + u[N-1][i];
   }
   mean /= (4.0 * N);
/* Initialize interior values */
   for (i = 1; i < N-1; i++)
      for (j = 1; j < N-1; j++) u[i][j] = mean;
```

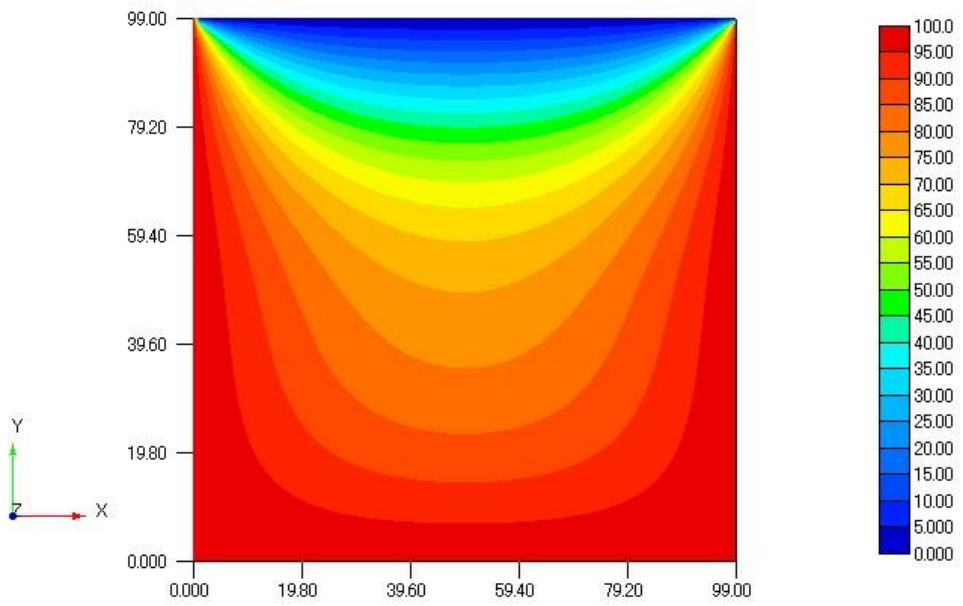
```
 /* Compute steady-state solution */
    for (;;) {
       diff = 0.0;
#pragma omp parallel private (i,j,tdiff)
{
       tdiff = 0.0;
#pragma omp for
       for (i = 1; i < N-1; i++)
          for (j = 1; j < N-1; j++) {
             w[i][j] = (u[i-1][j] + u[i+1][j] +
                        u[i][j-1] + u[i][j+1])/4.0;
             if (fabs(w[i][j] - u[i][j]) > tdiff)
             tdiff = fabs(w[i][j]-u[i][j]);
                        }
#pragma omp for nowait
       for (i = 1; i < N-1; i++)
          for (j = 1; j < N-1; j++) u[i][j] = w[i][j];
#pragma omp critical
             if (tdiff > diff) diff = tdiff;
}
    if (diff <= EPSILON) break;
    } /* Print solution */
    for (i = 0; i < N; i++) {
       for (j = 0; j < N; j++)
          printf ("%6.2f ",u[i][j]);
          putchar ('¥n');
    }
}
```

熱伝導方程式 –ポアソン方程式

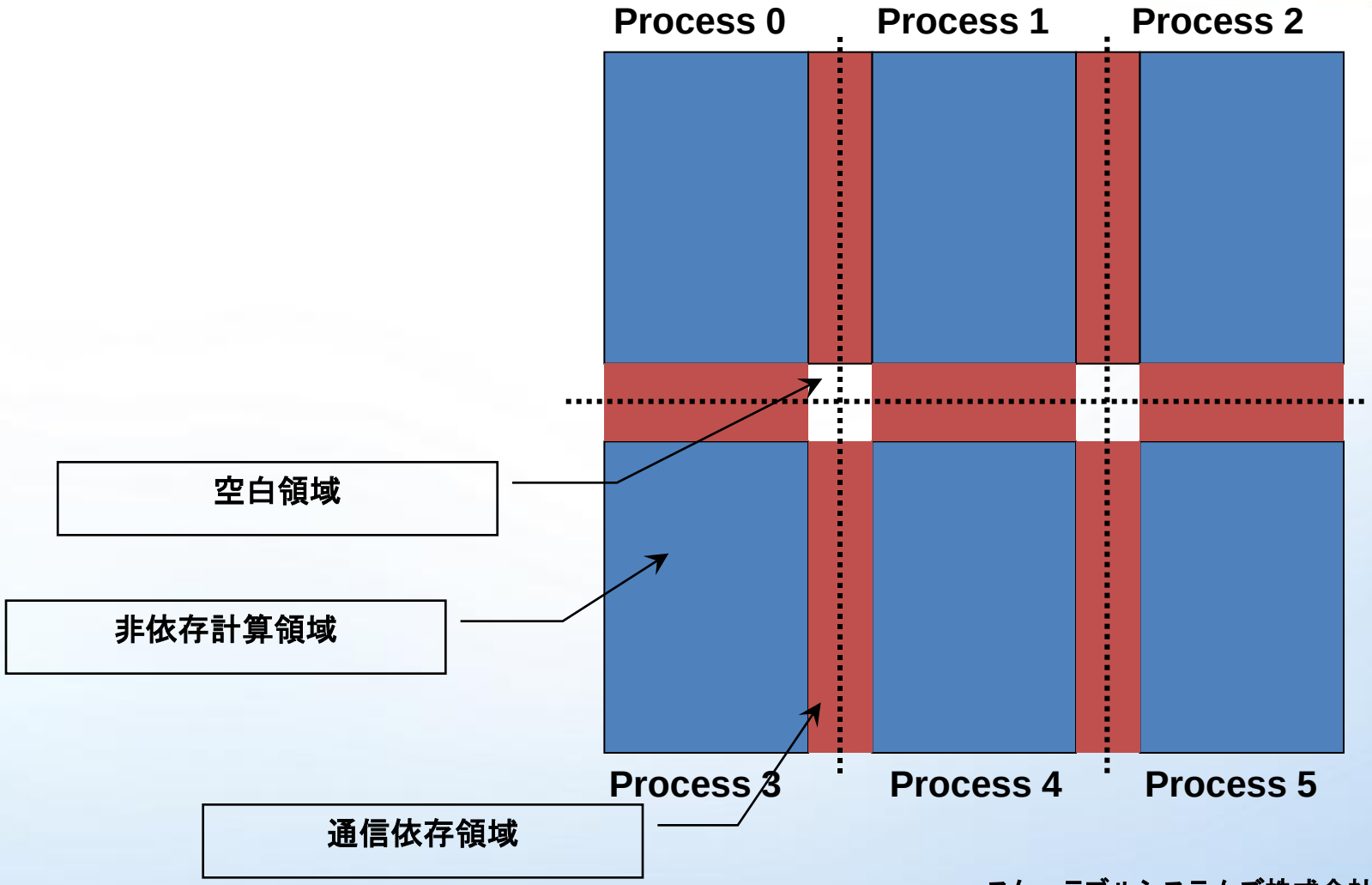
初期条件



計算結果



MPIでの領域分割


Process 0
Process 1
Process 2
空白領域
非依存計算領域
通信依存領域
Process 3
Process 4
Process 5

アグロメレーション（結合）

- 次のようにプリミティブ・タスクをグループ化

- パフォーマンス/粒度を向上
- 通信を集中
  - 通信するタスクを同じグループにする
- 設計上のスケーラビリティを維持
  - データセットのサイズまたはプロセッサ数による変更を細かく制御
- プログラミングとメンテナンスを単純化

作業のレプリケーション

- 通信を減らすための演算の複製とのトレードオフ
- どちらのタスクのアグロメレーションの方が同期が減るか?

順次アルゴリズム:

```
Compute X
Compute Y
Compute Z
For i = 1, 20
Compute a[i] = F(X,Y,Z,b[i])
```

A

スレッド 1

```
Compute X
Compute Y
Compute Z
Signal X,Y,Z ready
For i = 1, 10
Compute a[i] =F(X,Y,Z,b[i])
```

スレッド 2

```
Wait for  X,Y,Z ready


For i = 11, 20
Compute a[i] = F(X,Y,Z,b[i])
```

メモリのトレードオフに注意!

B

スレッド 1

```
Compute X
Compute Y
Compute Z
For i = 1, 10
Compute a[i] = F(X,Y,Z,b[i])
```

スレッド 2

```
Compute X
Compute Y
Compute Z
For i = 11, 20
Compute a[i] = F(X,Y,Z,b[i])
```

通信

- 並列プログラムにあって、逐次プログラムにないものが通信
- （共有メモリでの並列プログラミングでは、通信を意識してプログラムを書くことは実際はないので、一概に並列プログラムとは言えないが….）
- 通信では、通信の際には誰に送るのか，誰から受け取るのかを特定することが必要
- 通信パターンの決定
  - 一対一通信　（ローカル）
  - 集団通信　（グローバル）
- メッセージの順序関係には依存関係がある
  - 同期と非同期

同期処理

分散メモリシステム

- 共有データを持たず、各プロセスが、独自のメモリ領域を持つ
- 従って、同期 = 通信となる
  - MPIにおいては、同期通信を行った場合、データ転送の終了まで、その実行を待つことになる
- データへのアクセス制御
  - あるプロセスが、他のノード上のa[i]のデータを必要とした場合、そのデータを転送し、その転送が終了するまで、計算を進めることはできない

共有メモリシステム

- 同期処理は非常に重要
- データへのアクセス制御
  - バリア同期
  - クリティカル・セクション
  - 共有メモリAPIでは、メモリ上のa[i]は、いつでもアクセス可能であるが、そのデータの更新時期やアクセスのための同期処理はユーザの責任となる

スレッド実行時の同期処理

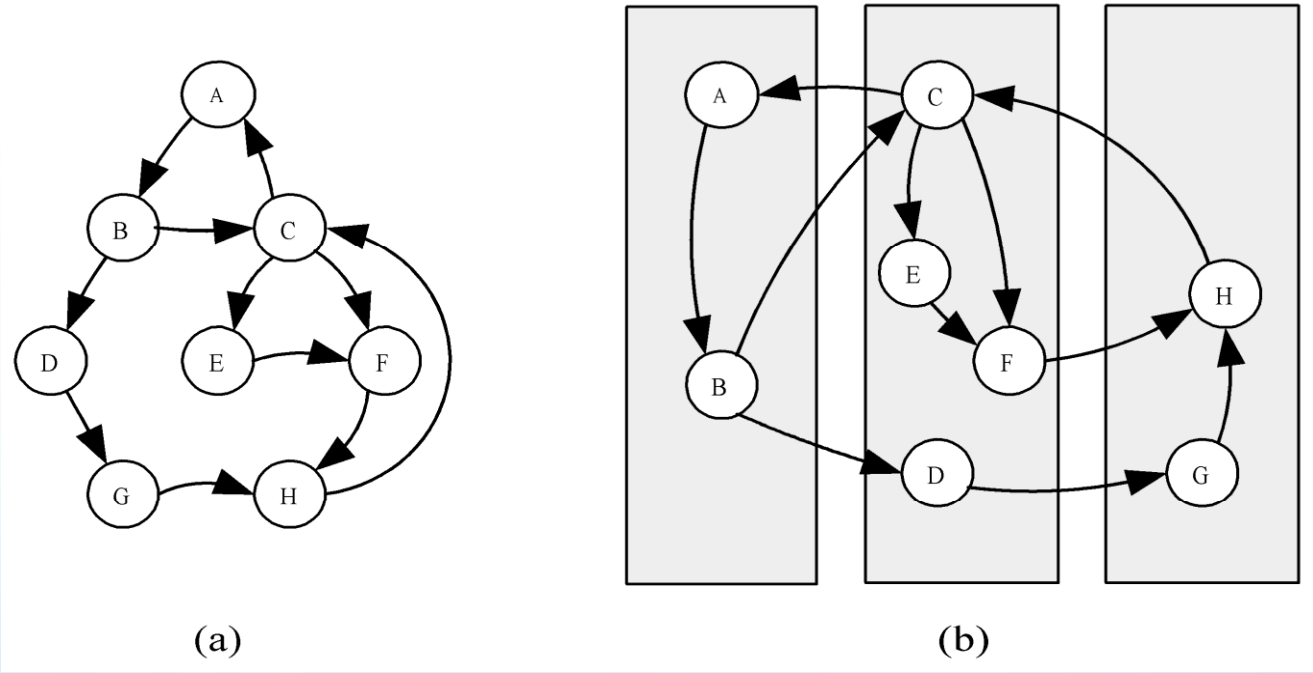
- スレッドでのプログラムの同期
  - タスク B の開始前にタスク A が終わることを保証する
- 同期処理機構
  - バリア
    - スレッドはすべてのスレッドがバリアに進むまで休止
  - イベント、シグナル、条件変数
    - スレッドは処理を進める前にシグナル（メッセージ）を待つ
  - クリティカル・セクション
    - アトミックに実行する必要があるコード・セクション
    - 割り込みなしで共有変数を読み取りまたは更新
  - ミューテックス

マッピング

- 次のようにプロセッサにタスクを割り当てる
  - プロセッサの使用率を最大化
  - プロセッサ間の通信を最小化
- プロセッサごとに 1 つのタスクか、複数のタスクか?
- 静的割り当てか、動的割り当てか?
- 大部分はメッセージ・パッシングに適用可能
  - 開発者はスレッドにタスクをマップできる

マッピング例

3つのプロセッサへのタスクの割り当てをここでは示しています。各タスクが同じ処理量（時間）だとすると、他の2つのプロセッサよりも一つのプロセッサの負荷がより多く（2倍）になっています。



スケジューリング

- メッセージ・パッシング
  - アプリケーションの最初にデータ/タスクを分割
  - 割り当てはスレッドの数に基づく
  - データの分散方法は?
    - 単一ソースからデータを送信
    - 単一プロセスで I/O を処理
    - 個別入力
- スレッド
  - OpenMPではスケジューリング制御用の実行時関数と環境変数をサポート
  - スレッドの場合、データの分散は基本的には不要で、データは共有メモリ領域に格納

スレッドプール

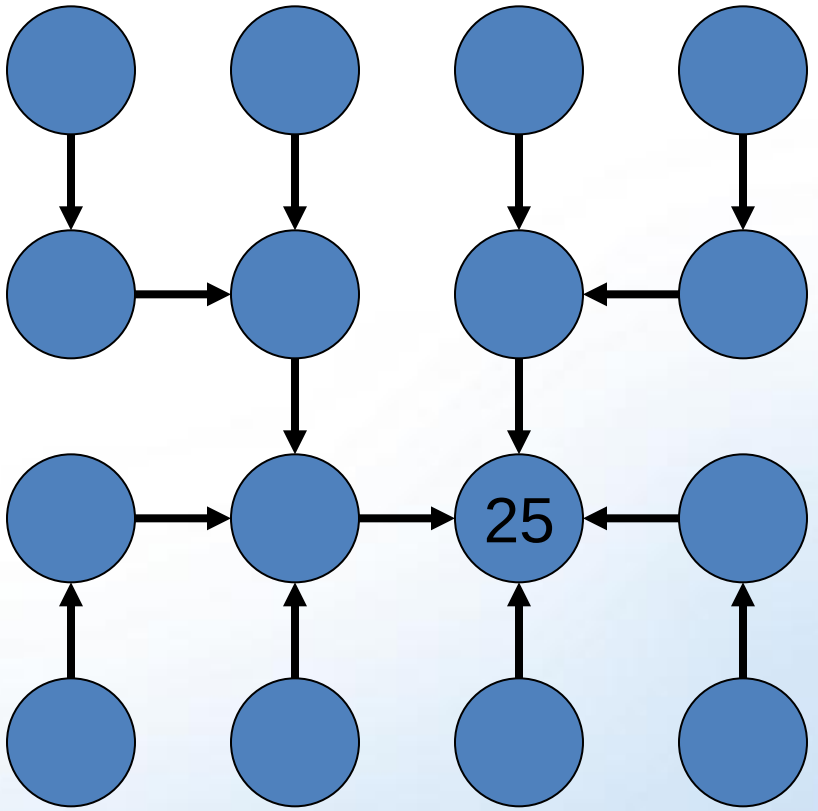
- Boss-Workerモデルでの並列処理
  - 小さなトランザクションを処理するアプリケーションに最適
  - 新しいトランザクションを処理するたびに「一時的な」スレッドを作成するのは非効率
    - スレッド作成と破棄のオーバーヘッド
- より良いソリューション: スレッドプール
  - スレッドの数を制限してスポーン
  - 制御するスレッドのトランザクションをキューに入れる
- インテルはOpenMP WorkQueueをサポート
- MPIでも実装可能

パーティショニングのチェックリスト

- 分割の品質の評価
- プロセッサ数よりもプリミティブ・タスクの数で概算されたか?
- 冗長演算およびデータ格納領域は最小限にされたか?
- プリミティブ・タスクはほぼ同じサイズか?
- タスクの数は問題箇所のサイズに基づいているか?

通信のチェックリスト

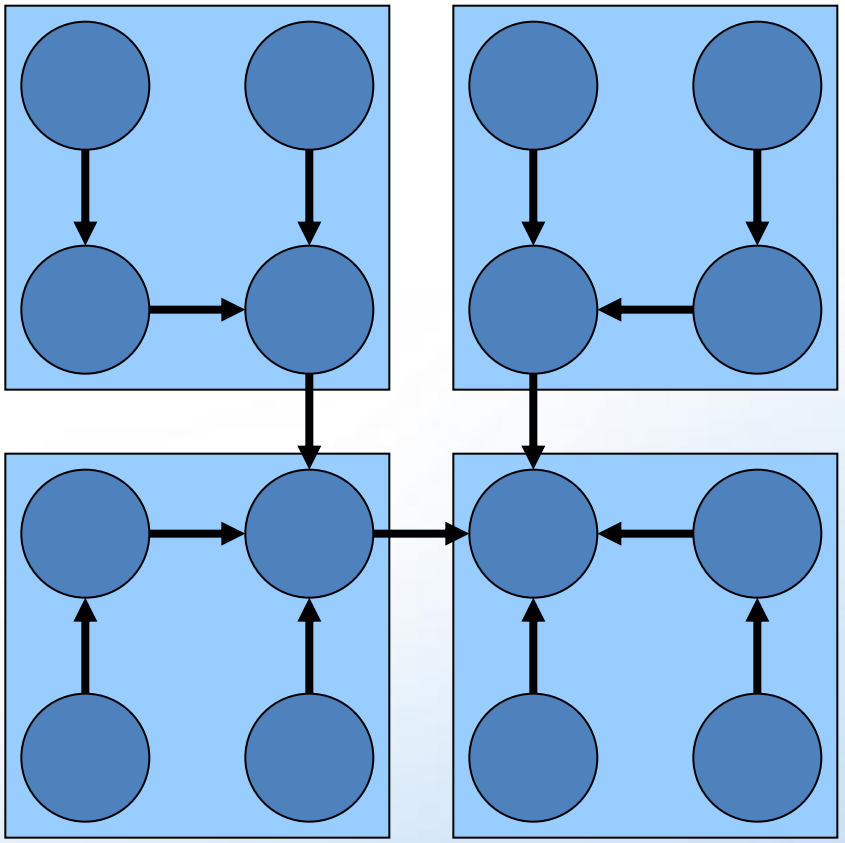
- 通信の品質の評価
- 通信操作はバランスが取れているか?
- 各タスクは少数の隣のタスクと通信しているか?
- タスクは通信を同時に実行できるか?
- タスクは演算を同時に実行できるか?



総和の計算時の通信パターン

アグロメレーションのチェックリスト

- アグロメレーションの品質の評価
- 通信の局所性は増加したか?
- 結合されたタスクの演算と通信は似ているか?
- 複製された演算は置換された通信よりも時間がかからないか?
- 複製されたデータの量はアルゴリズムがスケーリング可能な量か?
- コード修正のトレードオフは適切か?



マッピングのチェックリスト

- マッピングの品質の評価
  - プロセッサ設計について 1つのタスクと複数のタスクの両方が考慮されたか?
  - 静的割り当てと動的割り当ての両方が評価されたか?
  - 動的の場合、マネージャ・スレッドがボトルネックではないか?
  - 静的の場合、ロードバランスが考慮されたか?

リソース

- Foster, Ian T.著『Designing and Building Parallel Programs』Boston: Addison-Wesley, 1995
  - この本の内容は www-unix.mcs.anl.gov/dbpp から無料でダウンロード可能

並列化の阻害要因

- “ステート” を伴うサブプログラム
  - 擬似乱数生成
  - ファイル I/O ルーチン
- 依存関係があるループ
  - ある反復で書き込まれ、別の反復で読み取られる変数
  - ループキャリー：値をある反復から次の反復に運ぶ
  - 帰納変数：ループごとにインクリメントされる
  - リダクション： 配列を単一データに変換する
  - 循環：次の反復に情報を伝える

Thread-Safe

- 多くのルーチンは呼び出しの状態を維持する
  - メモリ割り当て
  - 擬似乱数生成
  - I/O ルーチン
  - グラフィック・ライブラリ
  - サードパーティ・ライブラリ
- これらのルーチンへの並列アクセスは同期されていない限り安全ではない（Thread-Safe)
- スレッドの安全性を決定する特定の関数についてのドキュメントを確認する

ループにおける反復間の依存関係

- 変数wrap が 1 つの反復から次の反復に依存性を持っているため、このループは並列ではない
  - 変数wrap が各反復で使用される前に定義されるように再構成する

```
wrap = a[0] * b[0];
for (i=1; i<N; i++) {
   c[i] = wrap;
   wrap = a[i] * b[i];
   d[i] = 2 * wrap;
}
```

```
for (i=1; i<N; i++) {
   wrap = a[i-1] * b[i-1];
   c[i] = wrap;
   wrap = a[i] * b[i];
   d[i] = 2 * wrap;
}
```

帰納変数

- 帰納変数は各ループの反復毎にインクリメントされる
  - インクリメント式をループ・インデックスから計算される関数に置き換える

```
i1 = 0
i2 = 0
DO I=1,N
   i1 = i1 + 1
   B(i1) = ...
   i2 = i2 + I
   A(i2) = ...
ENDDO
```

```
DO I=1,N
  B(I) = ...
  A((I**2 + I)/2)= ...
ENDDO
```

リダクション

- リダクションは、アソシエーティブ演算により配列データをスカラデータに変換する
- アソシエーティビティを利用して、プライベート領域の部分和または極大値を計算する
- 次に、アクセスが同期するように注意しながら、部分的な結果を共有結果と組み合わせる

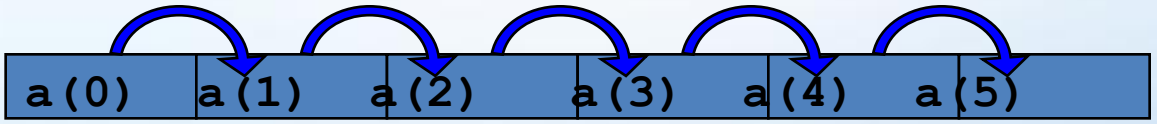
```
do i=1,n
   sum = sum + c(i)
   maxx = max(maxx,c(i))
enddo
```

```
for (i=0; i<n; i++)
    sum += c[i];
```

循環

- 循環関係は、ある反復から次の反復に情報を伝える
  - 時間ステップのループ
  - 収束ループ
- 大部分の循環は完全に並列化できない。代わりに、より外側のループまたはより内側のループを探す

```
do i=1,n
  a(i) = a(i-1) + b(i)
enddo
```



並列プログラムにおける留意点

- プログラムの逐次実行では、発生しない問題が並列処理では発生する

- デッドロック
  - 決して発生しないイベント/オブジェクト/メッセージを待つスレッド
- 丸め誤差

デッドロック

メッセージ・パッシング

- 実行していないプロセスからのメッセージを待つ
- 不適切な送信と受信操作の組み合わせ
- 異なるバリアで待機

スレッド

- 正しくない階層のロック
- 同期オブジェクトの保持を終了したスレッド
- 異なるバリアで待機

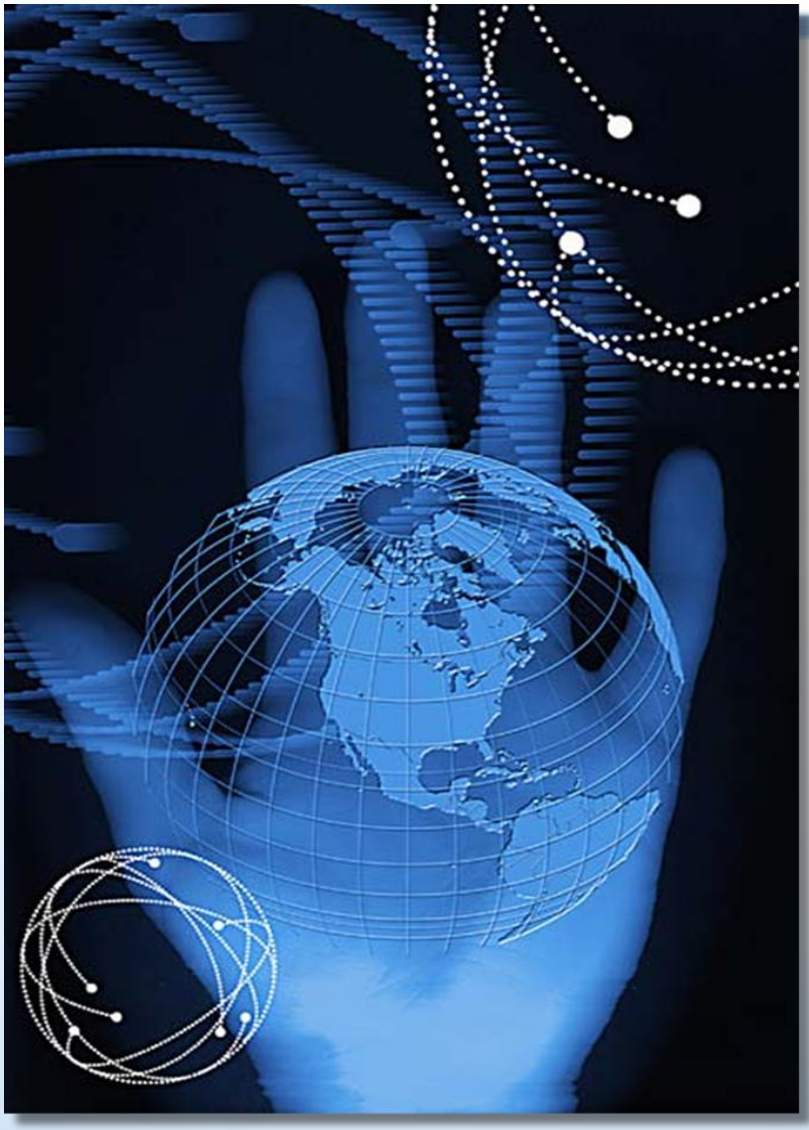
丸め誤差

- 有限桁数で行われるコンピュータ演算には誤差が含まれる
- 並列処理では、その演算順序が並列処理を行わない場合と変わる可能性がある
- 例：非常に小さな数値ε（1.0 + ε = 1.0 のような非常に小さな数）に対する浮動小数点演算では、以下のような結果となる可能性がある

$$-1.0 + (1.0 + \varepsilon) = -1.0 + 1.0 = 0.0$$

$$(-1.0 + 1.0) + \varepsilon = 0.0 + \varepsilon = \varepsilon$$

この資料について



お問い合わせ
0120-090715
携帯電話・PHSからは（有料）
03-5875-4718
9:00-18:00 (土日・祝日を除く)

WEBでのお問い合わせ
www.sstc.co.jp/contact

1/17/2010

スケーラブルシステムズ株式会社